Blueprints – Illustrating Architecture and Technical Parts using Hardware-Accelerated Non-Photorealistic Rendering

Marc Nienhaus Jürgen Döllner

University of Potsdam
Hasso Plattner Institute

Abstract

Outlining and enhancing visible and occluded features in drafts of architecture and technical parts are essential techniques to visualize complex aggregated objects and to illustrate position, layout, and relations of their components.

In this paper, we present blueprints, a novel non-photorealistic hardware-accelerated rendering technique that outlines visible and non-visible perceptually important edges of 3D objects. Our technique is based on the edge map algorithm and the depth peeling technique to extract these edges from arbitrary 3D scene geometry in depth-sorted order. After edge maps have been generated, they are composed in image space using depth sprites, which allow us to combine blueprints with further 3D scene contents. We introduce depth masking to dynamically adapt the number of rendering passes for highlighting and illustrating features of particular importance and their relation to the entire assembly. Finally, we give an example of blueprints that visualize and illustrate ancient architecture in the scope of cultural heritage.

Key words: Blueprints, non-photorealistic rendering, hardware-accelerated rendering, edge map, depth peeling.

1 Introduction

The term *blueprint* in its original meaning denotes "a photographic print in white on a bright blue ground or blue on a white ground used especially for copying maps, mechanical drawings, and architects' plans" (Merriam Webster). Blueprints consist of transparently rendered features, represented by their outlines. This way, blueprints allow for realizing complex, hierarchical object assemblies such as found in architectural drafts, technical illustrations, and design.

This paper introduces a general-purpose hardware-accelerated rendering technique for generating blueprints of arbitrary 3D object assemblies, outlining position, layout, and relations of each feature (Fig. 1). This way, blueprints become an effective visualization tool for interactively exploring complex objects and communicating structure and relationships of their components. Among the many application areas, blueprints can be used for visualizing and illustrating ancient architecture of cultural heritage as demonstrated.

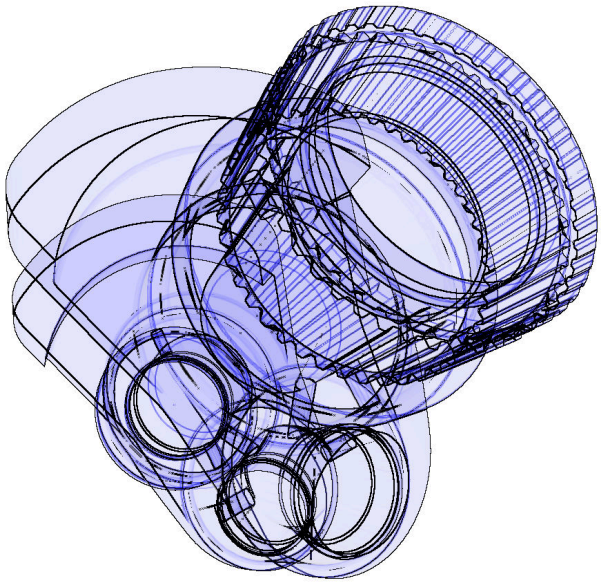

Figure 1: The blueprint of a crank generated with our technique outlines its design comprehensibly.

Our blueprint technique applies non-photorealistic rendering (NPR) to enhance perceptually important edges of 3D scene geometry, achieving vivid and expressive depictions and facilitating visual perception. It also applies depth peeling as a technique to decompose arbitrary 3D scene geometry in disjunctive depth layers to cope with its depth complexity. The blueprint technique operates in image space and takes fully advantage of hardware-acceleration, thus being applicable for real-time rendering.

In a naïve approach for blueprints, a wire-frame drawing could be used, but would not allow us to distinguish between triangulation edges and true outlines (e.g., silhouettes) and even complicate the visual perception of complex object assemblies. One could also use transparency rendering, but outlines would be hardly visible, in particular in regions of high depth complexity. Existing image-space NPR algorithms, finally, operate on

visible features and cannot be directly extended towards transparent renderings. Our approach extends and integrates an NPR edge enhancement technique with depth peeling.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 describes our implementation of depth peeling. Section 4 presents our blueprints rendering technique. Section 5 introduces applications, and Section 6 draws conclusions and discusses future work.

2 Related Work

Non-photorealistic rendering has become a popular research topic in computer graphics during the last decade. NPR includes rendering styles such as painterly rendering [10], hatching [21], and edge enhancement [18]. Today, a number of object space [4,11], image space [5,17,24], and hybrid [20] algorithms exist for detecting and enhancing visually important edges of 3D scene geometry. Outlining edges is an important technique to differentiate the parts of complex 3D objects; traditionally it is used in technical illustrations and visual instruction guides [1]. By their nature, image space techniques for edge enhancement are nearly independent from polygon count and impose few prerequisites on 3D scene geometry. Object space and hybrid approaches allow for generating stylized silhouettes; they can be applied to occluded edges as well [13]. Isenberg et al. [12] give an overview on silhouette extraction algorithms.

Increasingly, real-time non-photorealistic rendering algorithms are accelerated taking advantage of graphics hardware available today [8,21,22]. In particular, image space rendering has been accelerated, now being usable for real-time image processing operations even for non-photorealistic rendering [16,17].

One of the pioneering works in non-photorealistic rendering is the G-buffer concept introduced by Saito and Takahashi [24]. Geometric buffers are 2-dimensional data structures that store geometrical properties of 3D scene geometry. Important G-buffers are the normal buffer, the z-buffer, and the Id-buffer. Furthermore, image processing operations are provided with G-buffers to analyze their contents and to produce comprehensible images of 3D scene geometry. These include edge-enhanced or hatched renderings of 3D scene geometry.

Decaudin [5] introduces cartoon-style renderings of 3D scene geometry with enhanced edges. His method detects edges by extracting discontinuities using image processing techniques applied to the normal and z-buffers. His approach is not intended being processed in real-time.

Image processing techniques applied to 2-dimensional images are time consuming. However, Mitchell et al. [17] present a real-time rendering technique to extract edges in image space to enhance 3D scenes taking full advantage of hardware-acceleration on a per-scene basis. Their method renders fragment normals, z-values, and object identifiers of 3D scene geometry into textures using a render-to-texture implementation. Then, it detects discontinuities in these buffers using graphics hardware and combines the resulting edges with frame-buffer contents. This way, edges of 3D scene geometry, regions in shadow, and texture boundaries can be outlined.

An edge-enhancement algorithm that extracts discontinuities in G-buffers on a per-object basis using graphics hardware has been introduced in [18]. It distinguishes profile edges and edges of inner forms by handling discontinuities in the normal and z-buffer differently. The assembly of intensity values constituting edges is made available by a texture, called edge map. The algorithm preserves the edge map, so that it can be combined with manifold non-photorealistic rendering algorithms [8,9,21] and advanced multipass, real-time rendering algorithms [3].

The blueprint technique extends the edge map algorithm to generate edge maps for varying depth levels. These edge maps are used to compose blueprints in a subsequent rendering pass of our multipass rendering algorithm.

Mammen [15] implements a high-quality antialiased transparency rendering algorithm as an application of the Virtual Pixel Maps architecture. For it, he introduces a solution to incorporate processing pixels in depth-sorted order with the z-buffer concept. Thus, his multipass rendering algorithm generates an ordering of transparent pixels suitable for transparency rendering.

Diefenbach [6] uses the dual z-buffer to implement two depth tests for each fragment. This way, the additional z-buffer allows for rendering fragments in depth-sorted order and facilitates transparency rendering.

Everitt [7] introduces an alternative solution for order-independent transparency that is fully accelerated. To facilitate two depth tests without a dual z-buffer his technique uses shadow mapping hardware for projecting z-values back onto 3D scene geometry. This way, he implements depth peeling – a technique for extracting layers of ordered depth on a per-fragments basis. Each layer is captured as texture that is blended in depth-sorted order with frame buffer contents in the end.

The blueprint technique implements depth peeling to extract layers of ordered depth in image space. The general layout of its implementation allows us to integrate edge map construction for each layer easily. In a final rendering pass, we compose the blueprint by rendering edge maps as 2D textures as a kind of depth sprite. This way, we are even able to combine blueprints with arbitrary 3D scene geometry. Thus, blueprint rendering as a

```
procedure depthPeeling(G ← 3D scene geometry) begin
i=0
do
  F ← rasterize(G)
  if (i==0) begin
    ∀ fragment ∈ F begin
      bool test ← performDepthTest(fragment)
      if (test) begin
        fragment.depth → z-buffer
        fragment.color → color buffer
      end
      else reject fragment
    end
  end
  else begin
    ∀ fragment ∈ F begin
      if (fragment.depth > fragment.value depth layer map(i-1))
      begin
        bool test ← performDepthTest(fragment)
        if (test) begin
          fragment.depth → z-buffer
          fragment.color → color buffer
        end
        else reject fragment
      end
      else reject fragment
    end
  end
  depth layer map(i) ← capture(z-buffer)
  color layer map(i) ← capture(color buffer)
  i++
while (occlusionQuery(F) ≠ ∅ )   /* Condition */
end
```

Figure 2: Pseudocode illustrating our implementation of depth peeling.

tool can complement and enrich applications that visualize and communicate spatial relations [23], for instance, in applications that illustrate assembly instruction [1] and architecture [2].

3 Depth Peeling and Its Implementation

Depth peeling is operating on a per-fragment basis and allows for extracting 2D layers of 3D scene geometry in depth-sorted order. Generally speaking, depth peeling successively "peels away" layers of unique depth complexity.

In regular real-time 3D rendering, fragments passing an ordinary depth test define the minimal z-value at each pixel. But the fragment that comes second (or third, etc.) with respect to its depth cannot be determined. Thus, an additional depth test is needed to extract those fragments that form a layer of a given ordinal number with respect to depth.

Depth peeling, a multipass algorithm, allows us to step deeply into 3D scene geometry subject to the number of rendering passes while capturing each layer in a 2D texture. Thus, the first n layers are extracted by n rendering passes.

We refer a layer of unique depth complexity to as *depth layer* and a high-precision texture received from capturing the according z-buffer contents as *depth layer map*. The contents of the corresponding color buffer captured in an additional texture is called *color layer map*. Color layer maps can be used to compose the final rendition in depth-sorted order. For example, ordered blending each layer map in the frame buffer generates order-independent transparency [7].

The pseudocode in Figure 2 outlines our implementation of depth peeling. It operates on a set G of 3D scene geometries. G is rendered multiple times, whereby the rasterizer produces a set F of fragments. The loop terminates, if no fragment is rendered (termination condition), otherwise it continues to extract the next depth layer. Generally speaking, the condition is satisfied if the number of rendering passes has reached the maximum depth complexity.

In the first rendering pass (i=0) an ordinary depth test is performed on each fragment, thus, filling the z-buffer and the color buffer. Their contents are captured in a depth layer map resp. color layer map for further processing.

In consecutive rendering passes (i>0) an additional depth test is performed on each fragment. For it, the depth layer map produced in the previous rendering pass (i-1) serves for texture mapping 3D scene geometry. To do so, texture coordinates for a fragment are determined in such a way that they correspond to canvas coordinates of the targeted pixel position. This way, a texture access provides a fragment with the z-value stored at that pixel position in the z-buffer of the previous rendering pass. Now, the additional depth test works as follows:

- If the current z-value of a fragment is greater than the texture value that results from depth layer map access, the fragment proceeds and the following ordinary depth test is performed.

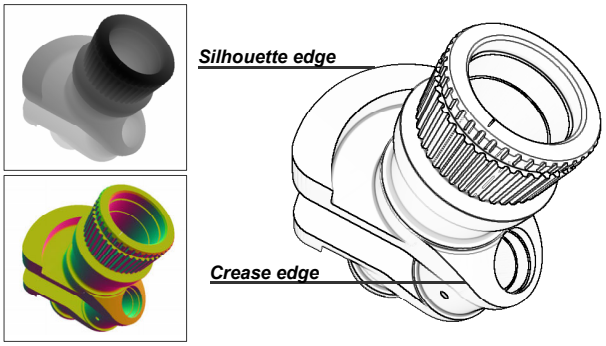

Figure 3: Discontinuities in the z-buffer and normal buffer form edge intensities in the edge map.

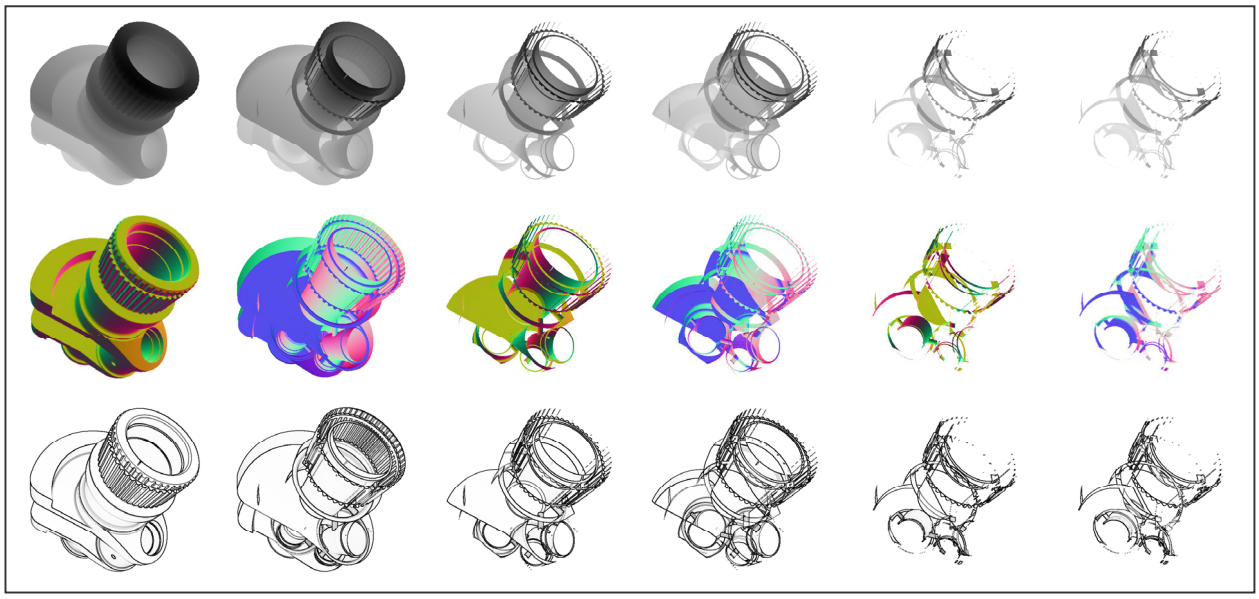

Figure 4: The z-buffer (first row) and the normal buffer (second row) of each depth layer (column) form basis for constructing the edge map (third row) for each layer.

- Otherwise, if the test fails, the fragment is rejected prior.

Again, after all fragments have been processed, the contents of the z-buffer and color buffer form a new depth layer map and color layer map. We can efficiently implement the additional depth test on a per-fragment basis using a fragment program. Furthermore, we utilize the occlusion query extension [14] to efficiently implement the termination condition.

4 Generating Blueprints

Our blueprint technique 1) extracts visible and non-visible edges of 3D scene geometry and 2) composes them as blueprints in the frame buffer.

4.1 Visible and Non-Visible Edges

Perceptually important edges include silhouette, border, and crease edges of 3D scene geometry. Our technique extracts these edges by determining discontinuities in both the normal and z-buffer. For it, encoded normals and z-values of 3D scene geometry are rendered directly into textures. So, as a prerequisite, 3D scene geometry must provide per-vertex normals. Then, a screen-aligned quad that fits completely into the viewport of the canvas is textured with these textures. Sampling neighboring texels allows for extracting discontinuities that result in intensity values constituting edges of 3D scene geometry. The assembly of edges forms a single texture that is called edge map. Figure 3 illustrates the normal buffer, the z-buffer, and the resulting edge map.

We classify visible and non-visible edges of 3D scene geometry as follows.

- Visible edges are edges directly seen by the virtual camera.
- Non-visible edges are edges that are occluded by faces of 3D scene geometry, i.e., they are not directly seen.

Our technique combines depth peeling with edge map construction to extract visible and non-visible edges of 3D scene geometry. Since visible edges are constituted by discontinuities in the normal buffer and z-buffer both have to be constructed. Encoding fragment normals as color values generates the normal buffer as color layer map for each rendering pass. Then, the edge map can be constructed directly since the depth map is already available. Non-visible edges become visible when depth layers are peeled away. This way, non-visible edges can be extracted successively. Thus, we complement our depth peeling implementation by constructing an edge map for each depth layer. As a result, the technique preserves visible and non-visible edges for further processing.

Figure 4 shows z-buffers, normal buffers, and resulting edge maps of several successive depth layers. It can be observed that edges may appear repeatedly in edge maps of consecutive depth layers. This results from those discontinuities that remain local if faces of 3D scene geometry are peeled away. Consider the following cases:

1) Two polygons are connected and share the same edge. One polygon occludes the other one. The discontinuity in the z-buffer that is produced along the

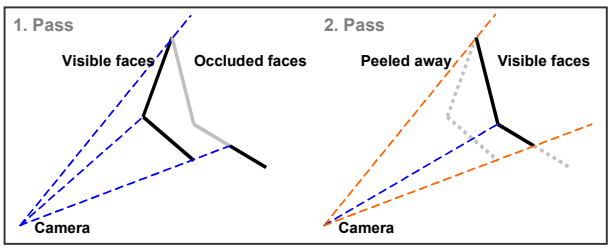

Figure 5: Top views of a set of upright polygons. Rays are cast to discontinuities that are produced by the composition of polygons and that are visible from the camera position. The left view illustrates the first rendering pass. The right view shows the same composition with faces peeled away. Note that the orange rays indicate edges that exist in both edge maps.

shared edge will remain if the occluding polygon is peeled away.

2) A polygon that partially occludes another polygon produces discontinuities in the z-buffer at the transition. If the occluding polygon and non-occluded portions are peeled away, a discontinuity in the z-buffer will be produced at the same location.

Figure 5 illustrates both cases. However, the performance of edge map construction is independent of the number discontinuities.

4.2 Composing Edge Maps

The blueprint of 3D scene geometry is composed by depth sprite rendering using visible and non-visible edges, which are stored in edge maps in depth-sorted order.

Depth sprites are 2-dimensional images that provide an additional z-value at each pixel for depth testing. Depth sprite rendering is implemented based on fragment programming. For each edge map, we proceed as follows:

1) A screen-aligned quad that fits completely into the viewport of the canvas is textured with the edge map and the corresponding depth map.
2) The fragment program replaces fragment z-values by texture values derived from accessing the depth map. If the z-value equals 1 – which denotes the depth of the back clipping plane – the program rejects the fragment. Otherwise, the fragment's RGBA values are calculated using texture values derived from edge map access. Then, the fragment proceeds to the ordinary depth test.

For providing depth cue our technique uses color blending by considering intensity values derived from the edge map as blending factors. For an example, see Figure 1.

As a variation, we can derive blueprints in a wire-frame style. For it, we define a threshold value and reject fragments if the intensity value is above the threshold. Otherwise, fragments represent edges and are blended into the color buffer. Note, that depth sprite rendering facilitates composing blueprints with further 3D scene geometry in arbitrary order.

Everitt [7] has already observed that it is sufficient to blend just the first few color layer maps to compose transparency. The remaining layer maps could have less visually impact to the overall composition because only a few (often isolated) pixels are produced. Thus,

- restricting the number of rendering passes to a maximum, or
- specifying a desired minimal number of fragments (dependent on the window resolution) to pass the depth test

represent alternative termination conditions, which optimize rendering speed. We opt for the second choice, which decreases the number of rendering passes while maintaining visual quality of blueprints. To implement the trade-off between speed and quality, the occlusion query extension can be configured appropriately.

4.3 Depth-Masking Hidden Components

Outlining the area surrounding possibly occluded components or locations is of particular importance to understand their relation to the overall structure. Furthermore, extra highlighting can be used to focus attention.

For it, we introduce depth masking to peel away a minimal number of depth layers until a specified fraction of the occluded components becomes visible. Thus,

```
procedure depthPeeling(G , depth mask) begin
i=0
do
   F ← rasterize(G)
   if (i==0) begin
      ∀ fragment ∈ F begin
        …
      end
   end
   else begin
      ∀ fragment ∈ F begin
         if (fragment.depth > fragment.value_depth layer map(i-1))
         …
      end
   end
   depth layer map(i) ← capture(z-buffer)
   color layer map(i) ← capture(color buffer)  /* normals */
   edge map(i) ← edges(depth layer map(i),color layer map(i))
   i++

   quad ← createTexturedScreenAlignedQuad(depth mask)
   Q ← rasterize(quad)
   R ← passedDepthTest(quad)
while ( #R < fraction(#Q) )      /* Condition */
end
```

Figure 6: Pseudocode illustrating the modified depth peeling technique supporting edge map construction and depth masking.

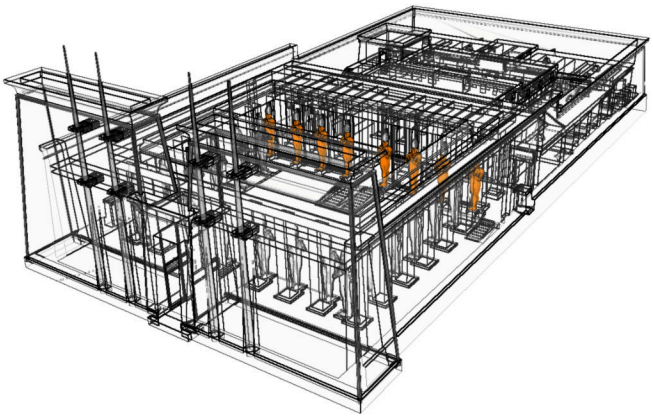

Figure 7: In the main building of the Temple of Ramses II statues guide the doorway from the inner yard to the rear chambers. Depth masking allows for outlining the structural design up to the highlighted statues.

depth masking provides a termination condition for blueprint rendering to dynamically adapt the number of rendering passes.

Depth masking works as follows:

1) An additional rendering pass is applied to generate a depth texture of designated components.
2) In successive rendering passes, our technique masks these components using the depth texture. To do so, a screen-aligned quad is rendered as depth sprite whenever a depth layer has been peeled away. If at least a specified fraction of fragments passes the ordinary depth test (based on z-buffer contents just produced), our technique terminates. Otherwise, further depth layers must be peeled away.
3) When composing blueprints, designated components are simply integrated.

The modifications to our technique are show in Figure 6. Again, we implement it using fragment programming and the occlusion query extension.

Figure 7 outlines the whole design of the entrance and the inner yard of the Temple of Ramses II with its surrounding walls and statues. These are in front of the highlighted statues that guard the doorway to the rear part of the temple. The number of depth layers that occlude the guarding statues and, therefore, have to be peeled away is determined by depth masking. Note that the full complexity of the rear part is not outlined.

4.4 Performance Remarks

The model of the crank (Fig. 1), which contains 25.000 triangles can be rendered at 5 fps at a window resolution of 512×512 using an NVidia GeForce FX 5600. Thereby, the 5 depth layers have been considered. Notably, this performance is almost independent from the CPU.

5 Applications

Visualizing and exploring architecture models is one key application area for blueprints, which is illustrated for ancient architecture in the following.

5.1 Plan Views Illustrating Architecture

Blueprints can be used to generate plan views to outline architecture comprehensibly. Composing plan views using an orthographic camera for blueprint rendering is a straightforward task.

Perceptual important edges are suitable to distinguish single components from each other in an overall composition. So, in a visualization of ancient architecture outlining the external and internal structure allows for identifying chambers, pillars, and statues systematically. Thus, blueprints increase visual perception in these illustrations. The plan views of the 3D model of the *Temple of Ramses II in Abydos* in Figure 8 are produced automatically.

5.2 Illustrating and Discovering Locations

Illustrating ancient architecture using glyphs allows for discovering and focusing on hidden details, locations, and relations that otherwise wouldn't have been noticed. Thus, combining general 3D scene geometry with blueprints can provide additional knowledge in depictions of archeology.

The illustrations in Figure 9 mark a hidden chamber

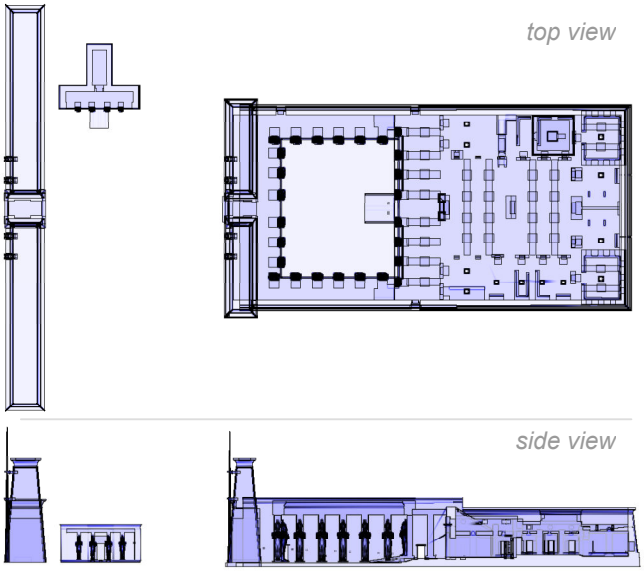

Figure 8: Blueprints illustrate a top and a side view of the Temple of Ramses II and outline its architectural design comprehensibly.

(red box) in the rear part of the Temple of Ramses II. Furthermore, the paths from the inner yard resp. outer side of the temple to the chamber are visualized (red arrows). The depth masking technique is used to peel away a minimal number of depth layers that hide the upper chamber and the pathways to it.

6 Conclusion and Future Work

We have presented blueprints, a non-photorealistic rendering technique for visualizing, illustrating, and outlining architecture and technical parts and its hardware-accelerated implementation.

We observed that orthographic views become more appropriate than perspective views with increasing structural complexity. In such cases, composing edges of consecutive depth layers could lessen comprehensibility; for instance, the rear chambers and the peristyle in Figure 8 can hardly be identified. Nevertheless, a perspective view still provides better spatial orientation and conceptual insight in blueprints. Therefore, future research should concentrate on techniques, such as depth masking, to determine those depth layers that contribute to comprehensible depictions. To implement our blueprint rendering technique we utilize the depth peeling technique and combine it with our edge map rendering algorithm. Actually, our technique takes full advantage of graphics hardware fragment programming and texturing capabilities. In addition, it scales with graphics hardware.

Future work will focus on accelerating our technique by utilizing upcoming concepts such as Super Buffers, which avoid context switches when using render-to-texture capabilities. Furthermore, we are going to integrate blueprints into an interactive visualization environment for 3D artifacts of cultural heritage.

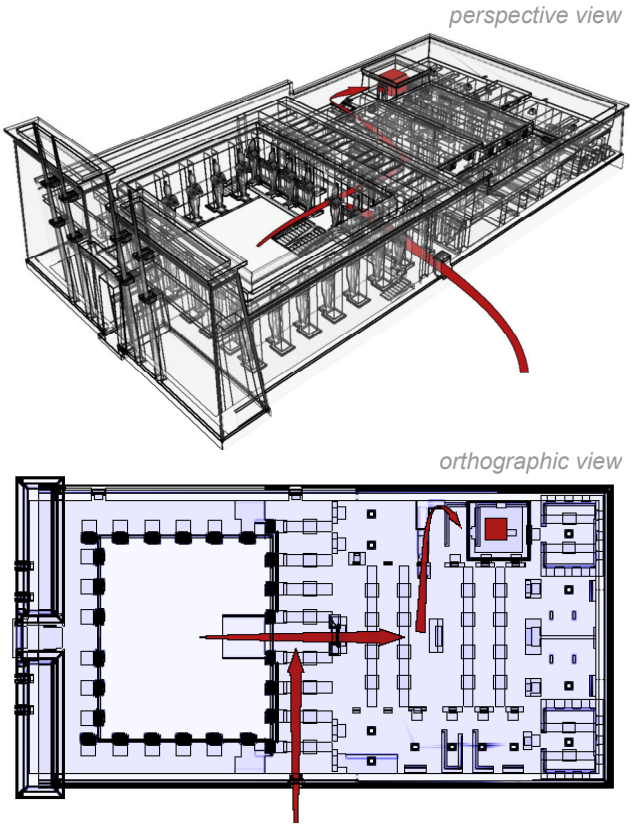

Figure 9: Blueprints enhanced with dynamic glyphs [19] illustrate paths through the arcades and the hall of the Temple to an upper chamber. The perspective view provides a spatial outline while the orthographic view provides a more systematic insight.

Acknowledgements

We thank ART+COM, Berlin for providing us the temple model and Bruce Gooch for the crank model. This work was supported by the 6. Framework Programme of the European Community within the TNT project FP6-2002-IST-1.

References

[1] M. Agrawala, D. Phan, J. Heiser, J. Haymaker, J. Klingner, P. Hanrahan, and B. Tversky. Designing Effective Step-By-Step Assembly Instructions. In *Proceedings of ACM SIGGRAPH 2003*, pp. 828-837, ACM Press, 2003.

[2] @Last Software, Inc. SketchUp, http://sketchup.com, 2002.

[3] D. Blythe, B. Grantham, M. J. Kilgard, T. McReynolds, and S. R. Nelson. Advanced Graphics Programming Techniques Using OpenGL. In *ACM SIGGRAPH 1999 Course Notes*, 1999.

[4] J. W. Buchanan and M. C. Sousa. The Edge Buffer: A Data Structure for easy Silhouette Rendering. In *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering*, pp. 39-42, Annecy, France, ACM Press, 2000.

[5] P. Decaudin. Rendu de scénes 3D imitant le style «dessin animé», *Rapport de Recherche 2919*, Université de Technologie de Compiègne, France, 1996.

[6] P. J. Diefenbach. Pipeline Rendering: Interaction and Realism Through Hardware-based Multi-Pass

Rendering. *Ph.D. thesis*, University of Pennsylvania, June 1996.

[7] C. Everitt. Interactive order-independent transparency. *Technical report*, NVIDIA Corporation, May 2001, Available at http://developer.nvidia.com.

[8] B. Freudenberg, M. Masuch, and T. Strothotte. Real-Time Halftoning: A Primitive For Non-Photorealistic Shading. In *Proceedings of 13th Eurographics Workshop on Rendering*, pp. 227-232, Pisa, Italy, Eurographics Association, 2002.

[9] A. Gooch, B. Gooch, P. Shirly, and E. Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. In *Proceedings of ACM SIGGRAPH 1998*, pp. 447-452, ACM Press, 1998.

[10] A. Hertzmann. Painterly Rendering with Curved Brush Strokes of Multiple Sizes. In *Proceedings of ACM SIGGRAPH 1998*, pp. 453-460, ACM Press, 1998.

[11] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *Proceedings of SIGGRAPH 2000*, pp. 517-526, ACM Press, 2000.

[12] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strotthotte. A Developer's Guide to Silhouette Algorithms for Polygonal Models. In *IEEE Computer Graphics and Applications*, 23(4), pp. 28-37, IEEE Computer Society Press, 2003.

[13] R. D. Kalnins, P. L. Davidson, L. Markosian, and A. Finkelstein. Coherent Stylized Silhouettes. In *Proceedings of ACM SIGGRAPH 2003*, pp. 856-861, ACM Press, 2003.

[14] M. Kilgard (Ed.), NVIDIA OpenGL Extension Specifications. NVIDIA Corporation, June 2003, Available at http://developer.nvidia.com/docs/IO/1174/ATT/-nvOpenGLspecs.pdf.

[15] A. Mammen. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. In *IEEE Computer Graphics and Applications*, 9(4), pp. 43-55, IEEE Computer Society Press, 1989.

[16] J. L. Mitchell. Real-Time 3D Scene Post-processing. *Game Developers Conference*, 2003, Available at http://www.ati.com/developer/gdc/GDC2003_-ScenePostprocessing.pdf.

[17] J. L. Mitchell, C. Brennan, and D. Card. Real-Time Image Space Outlining for Non-Photorealistic Rendering. In *ACM SIGGRAPH 2002 Conference Abstracts and Applications*, p. 239, ACM Press, 2002.

[18] M. Nienhaus and J. Döllner. Edge-Enhancement – An Algorithm for Real-Time Non-Photorealistic Rendering. In *Journal of WSCG*, 11(2), pp. 346-353, Plzen, 2003.

[19] M. Nienhaus and J. Döllner. Dynamic Glyphs – Depicting Dynamics in Images of 3D Scenes. In *Proceedings of the Third International Symposium on Smart Graphics,* pp. 102-111, Springer, 2003.

[20] J. D. Northrup and L. Markosian. Artistic Silhouettes: A Hybrid Approach. In *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering*, pp. 31-38, Annecy, France, ACM Press, 2000.

[21] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-Time Hatching. In *Proceedings of ACM SIGGRAPH 2001*, pp. 581-586, ACM Press, 2001.

[22] R. Raskar. Hardware Support for Non-photorealistic Rendering. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 41-46, ACM Press, 2001.

[23] F. Ritter, B. Preim, O. Deussen, and T. Strothotte. Using a 3D Puzzle as a Metaphor for Learning Spatial Relations. In *Proceedings of Graphics Interface 2000*, pp. 171-178, Morgan Kaufmann, 2000.

[24] T. Saito and T. Takahashi. Comprehensible Rendering of 3-D Shapes. In *Proceedings of SIGGRAPH 1990*, pp. 197-206, ACM Press, 1990.